ORA TM-96-0019 12 April 1996

Final Report

David Guaspari

N00014-95-C-0349
CDRL A002

Prepared for:

Program Manager, Ralph F. Wachter
ONR 311
800 North Quincy Street
Arlington, VA 2221705660

Prepared by:

David Guaspari
Odyssey Research Associates, Inc.
301 Dates Drive
Ithaca, NY 14850-1326
(607) 277-2020

1 Introduction

Formal methods research has developed a variety of mathematical tools and techniques applicable to the development of software systems, but they are greatly underused. The reasons, as documented in a careful survey of industrial applications of formal methods [2], include

- the limited mathematical backgrounds of many end-users and developers;
- support tools that are not of professional quality—fragile prototypes with poor interfaces and idiosyncratic notations;
- lack of attention to technology transfer—in particular, to reducing the risks and increasing the rewards of introducing formal methods techniques.

Different tools require different degrees of training and skill, and applying them requires different investments in time and effort. A large payoff would result from a formal methods interface (FMI) enabling different kinds of users, with different degrees of expertise, to cooperate in applying formal methods to define, explore, and analyze system designs and specifications. The payoff can be increased by exploiting the formal analysis for other purposes as well—for example, by generating documentation and code directly from formal models.

Devising an FMI therefore requires more than a technique for putting a modern interface, however good, on an interactive theorem prover. In the near- and medium-term, the greatest practical benefits will result from increasing the use of special purpose "low-end" or "lightweight" formal analysis through an interface that shields users from a Babel of differing notations and semantic models.

1.1 Lightweight analysis

Here is the form of lightweight analysis that our prototype FMI provides: Systems will be modeled in RSML, a mixed graphical and textual notation for defining state machines. Experience has shown that the RSML notation is accessible to end-users, engineers, and software developers. (There is a role here for expert consultants in how to read and write such specifications.)

The FMI provides a pushbutton way to select semantic questions about an RSML specification from predefined menu—e.g., the question of whether

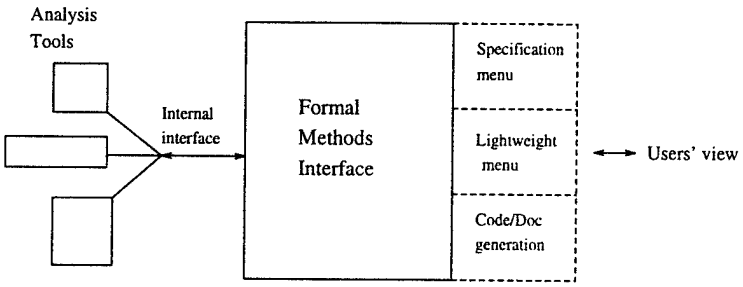

Figure 1: The lightweight FMI

the specified state machine can respond to every input. Our prototype generates a representation of each question within the formalism of various formal analysis tools (the EVES and PVS theorem provers, and the SPIN model checker) and permits the user to invoke those formal tools in an attempt to answer it automatically.

Integrating a tool with the FMI means providing, or automatically generating, some combination of theories, tactics, heuristics, decision procedures, etc., that give the tool a reasonable chance of answering some or all of the questions automatically. The result of the automated analysis may be a simple yes or no; or it may rephrase the problem in simpler terms that can be the basis for informal engineering judgements. If the automated analysis is insufficient, a more expert user of the analysis tools can apply them directly, in interactive mode. (There is a role here for consultants expert in using particular tools.)

Our goal is adapt tools to users (instead of the reverse)—reducing the entry costs and increasing the payoff from applying formal methods.

Figure 1 shows the role of the FMI as mediator between users and analysis tools, presenting both a unified view of the system under development and the ability to study that system directly within the formalism of one of the component tools.

1.2 Expert analysis

Phase I work has concentrated on a framework for lightweight analysis. We can also use the FMI to access more powerful features of the analysis tools. The state machine model provides a common semantic framework through

which different analysis tools can share their results. The FMI can, in effect, manage a theorem-proving session for a specialized top-level assertion language about our form of state machine model: invoking an analysis tool amounts to carrying out a special kind of proof step in that theorem prover. The FMI would maintain a database (not explicitly represented in figure 1) keeping track not only of the facts established, but of the justification for each fact. This database would help the user visualize the logical organization of the analysis, and help reestablish the consistency of the analysis after changes. Such a database would be integrated with ordinary project management tools that do workflow scheduling, configuration control, etc.

Research on these extensions is deferred to to Phase II.

1.3 Relation to other FMI projects

Our work on the underlying logic of an FMI is complementary to attempts to improve direct interactions with theorem provers and model checkers—and thus may well be complementary to other Phase I FMI projects.

1.4 Summary of results

In Phase I, we have accomplished the following:

We have chosen RSML (the Requirements State Machine Language) as a first draft of our state machine model and, in collaboration with the designers of RSML, have developed a more precise description of its semantics. Some of the remaining semantic questions and some further semantic developments are discussed in section 9.

We have devised general strategies for encoding the semantics of RSML features in formal analysis tools. We have prototyped code for a "logical interface" that made it possible to model these features rapidly in the EVES and PVS theorem provers and the SPIN model-checker. This code was developed by customizing certain predefined templates.

Little effort has been devoted to applying the prototype models, but we have have learned some things about the possible benefits of integrating each of the tools, their limitations, and developing a complete tool that would exploit them more effectively.

1.5 Organization of this report

This draft report is organized as follows:

- Section 2 describes the top-level notation, RSML.
- Section 3 describes RSML semantics and the lightweight analysis problems we are concerned with in Phase I.
- Section 4 describes the notion of a logical interface with RSML—essentially, a general strategy for modeling RSML concepts within typical analysis tools.
- Section 5 describes the internal interface to RSML—the customizable template for integrating analysis tools.
- Section 6 summarizes the results of Phase I and describes some "minimal paths" for turning this exploratory work into a useful product.
- Section 7 shows a sample encoding of a fragment of RSML semantics into an EVES theory.
- Section 8 shows a sample encoding of a fragment of RSML semantics as a SPIN model.
- Section 9 discusses some further questions in RSML semantics.
- There are also two attachments: the prototype code and its documentation; a paper by Nancy Leveson and Mats Heimdahl about RSML semantics, "Completeness and consistency in hierarchical state-based requirements"

2 Top-level notation

We see little value in performing FMI experiments with a toy specification language, so our first requirement for a top-level notation is that it be applicable to the demands of real-world projects. We have considered three closely related formalisms for describing state machines: StateCharts [5], which pioneered the basic ideas of graphic notations for hierarchical state machines; and two of its descendants, the ROOM modeling language [11] and RSML [9]. All have been applied to substantial practical problems.

This section briefly explains our reasons for choosing RSML.

2.1 RSML

From our point of view the principal virtue of RSML (the Requirements State Machine Language) is its practical origin as a *lingua franca* among all parties to the design of the complex TCAS II system, the airborne system that provides collision-warnings to commercial aircraft. The interested parties included pilots, engineers, outside examiners, and software developers.

RSML specifications are representable in two interchangeable formats—as (human-readable) ascii text, and as hypertext that mixes graphics and tabular notations. Thea ascii representation can easily be exploited to provide a simple interface to other tools. Detailed choices in the design of the RSML notation—for example, the decision to represent propositional formulas in tabular format rather than more conventional logical notation—were guided by experiences with the TCAS II project, a source of "human factors" experience that this project can hardly hope to improve on.

RSML has a formally definable semantics, though some design choices are still open and published formal descriptions have not been completely precise. We have collaborated with Mats Heimdahl, of the University of Michigan, to improve the semantic description. (As noted in section 9, some semantic questions remain.) The design of RSML has devoted considerable thought to the trade-offs between expressiveness and analyzability, and to identifying properties of RSML specifications for which formal analysis seems tractable. We adopt (a subset of) these already-identified analysis questions as our initial list of menu options for the lightweight FMI.

Finally, it is easy to experiment with RSML because its developers have generously provided us with the source code for their prototype front-end.

The body of this report will describe features of RSML as needed. A detailed description and an account of its use on TCAS II can be found in [9], and we have included as an appendix a paper on the semantic analysis of RSML [8].

We leave open the possibility of extending RSML to exploit opportunities not considered in the original design—opportunities opened up by integrating automated theorem provers and model checkers. For example, RSML specifications define pure finite-state machines (with inputs and outputs). So, for example, a counter must be modeled not by an internal variable, but by adding an internal state for each possible counter value. Formulating invariants on such local variables is a powerful and well understood technique for specification and analysis, and the availability of theorem-provers and model-checkers provides an opportunity to automate some of that analysis.

2.2 ROOM and StateCharts

The ROOM (Real-time Object Oriented Modeling) language adds object-oriented features to the hierarchical state-machine notation of StateCharts. It is supported by a commercial tool, ObjecTime, aimed at developers of real-time systems. This tool provides simulation and code generation, as well as a highly professional interface in which large amounts of information (the communications interfaces between state-machine modules, the transitions of individual state-machines, the object hierarchy, etc.) are hyperlinked. The developers of ObjecTime generously allowed us to attend one of their week-long training sessions.

ROOM permits local variables that maintain state information (potentially a good thing) but also permits a model to define the effect of a state machine transition by the execution of an arbitrary piece of C++ code. From our point of view that is a fatal permission, as it leaves ROOM without a mathematically defined semantics. (The semantics is well-defined in the sense of being defined by the action of the ObjecTime simulator, but there is no mathematical definition of that behavior.)

StateCharts is supported by the commercial product StateMate, which provides simulation, code generation, and dynamic testing. StateCharts has a mathematical semantics. It provides a richer language than RSML and, as a result, static formal analysis seems likely to be more difficult. It seems fair to say that RSML is more narrowly aimed at safety-critical applications and at developers willing to accept a more constrained style of specification and implementation, while StateCharts is a more general-purpose language for modeling reactive systems.

3 RSML semantics

This section contains a brief description of the heart of RSML, the notions of hierarchical states and transitions.

3.1 Hierarchical states

The global state of an RSML machine consists of a history of its inputs, a history of its outputs, and a *configuration*. The configuration is a set of (local) states that is consistent with the state hierarchy.

Figure 2 contains shows a graphical representation that includes the state hierarchy and a collection of arrows representing local transitions be-

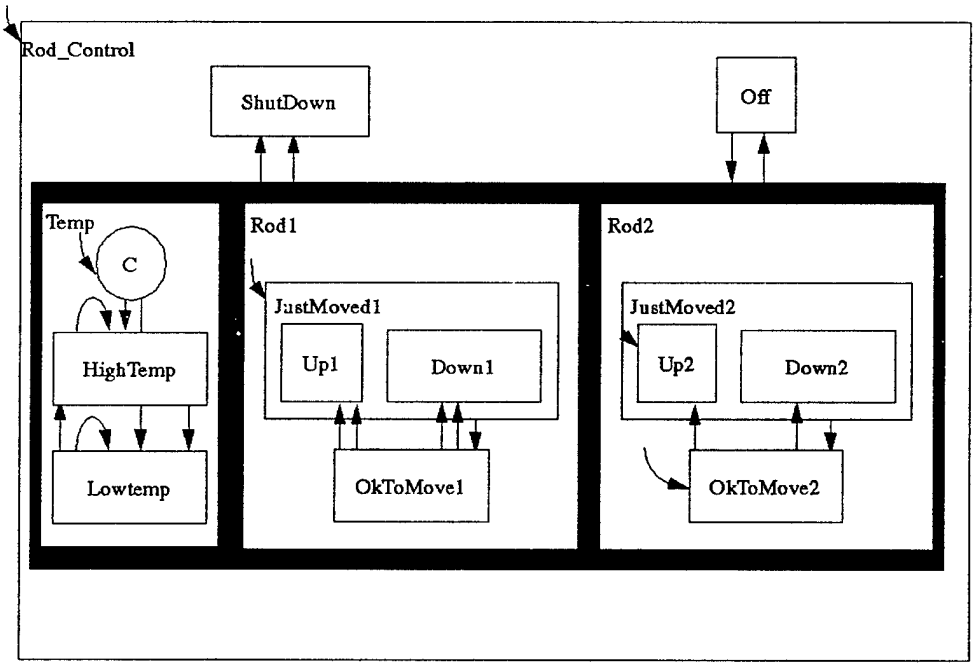

Figure 2: Graphical display of state machine

tween states. The text fragment describing this state hierarchy (though not describing the arrows) is given in figure 3.

In figures 2 and 3, Rod_Control, Off, Shutdown, On, Temp, etc., are the names of states. The top-level state, Rod_Control is an OR of its three children, Off, Shutdown, and On: to be in state Rod_Control is to be in precisely one of its children. State On is an AND of its children, Temp, Rod1, and Rod2: the children name concurrent processes and being in state On means being in all three of its child states simultaneously. State Rod1 is in turn an OR-state, so being in Rod1 means being in precisely one of the states JustMoved1 or OkToMove1. Etc.

Configurations are maximal consistent sets of states, for example:

{Rod_Control, Off}

{Rod_Control, On, Temp, HighTemp, Rod1, OkToMove1, Rod2, JustMoved2, Up2}

Graphically, AND-states are distinguished from OR-states by making the background of an OR-state clear and that of an AND-state grey. The significance of the DEFAULT children of an OR-state is explained below.

3.2 Basic transitions

Basic transitions are represented partly in graphical form (by the arrows in the figure, which indicate the source and destination of each transition) and partly in textual and tabular form (where the causes and the effects of each transition are described). Each basic transition has a single source state and a single destination state, and represents (among other things) a local change to the configuration. Transitions of the whole state machine are defined as combinations of basic transitions.

Thus, the arrow from source On to destination Off represents a transition that exits state On and all of its descendants, and enters state Off. The arrow from Off to On represents a transition that exits state Off and enters On, and here the defaults come into play. (A default state is indicated graphically by an arrow whose head touches the boundary of the state, and whose tail is not attached to any state.) For example, if we take the transition represented by the arrow from Off to On then we must also enter state Rod1, which in turn means that we must enter precisely one of its child states. Since the arrow stops at the boundary of On, it doesn't explicitly specify which child of Rod1 to choose; therefore we enter the default state JustMoved1. The default state can be defined conditionally, depending on external circumstances,

```
OR_STATE Rod_Control DEFAULT Off :
    ATOMIC Off :
    ATOMIC ShutDown :
    AND_STATE On :
        OR_STATE Temp DEFAULT C :
            CONDITIONAL C :
            ATOMIC HighTemp :
            ATOMIC LowTemp :
        END OR_STATE
        OR_STATE Rod1 DEFAULT JustMoved1 :
            OR_STATE JustMoved1 DEFAULT Up1 :
                ATOMIC Up1 :
                ATOMIC Down1 :
            END OR_STATE
            ATOMIC OkToMove1 :
        END OR_STATE
        OR_STATE Rod2 DEFAULT OkToMove2 :
            OR_STATE JustMoved2 DEFAULT Up2 :
                ATOMIC Up2 :
                ATOMIC Down2 :
            END OR_STATE
            ATOMIC OkToMove2 :
        END OR_STATE
    END AND_STATE
END OR_STATE
```

Figure 3: Textual display of state hierarchy

and that is the role of the conditional state C, but this detail is not relevant here. Section 9 discusses the somewhat problematic semantics of conditional states, and the problem of assigning a meaning to arrows whose head and tail are attached to states that are remote from one another, such as an arrow that led directly from OKToMove2 to Off.

There are two arrows from OkToMove1 to Up1. That is, there are two such basic transitions, with distinct names, which may have different causes and/or effects. The textual representations of these transitions (labeled rather arbitrarily t11 and t12) are:

```
TRANSITION t11 FROM OkToMove1 TO Up1 :
TRIGGER    : TempLow_Event
CONDITION  : TRUE
ACTION     : Rod1_Up
END TRANSITION

TRANSITION t12 FROM OkToMove1 TO Up1 :
TRIGGER    : TIMEOUT (TIME ( Rod1_Down ),
                              Temp_Reading_Timeout)
CONDITION :  TABLE
                TIME >= PrevTempLate()                    : T ;
END TABLE
ACTION     : Rod1_Up
END TRANSITION
```

To keep the picture from becoming too cluttered, this text is not displayed on the graph of states and arrows. Instead, its on-line version is available through a hyperlink active when the cursor is pointing to the transition.

The definition of a basic transition lists the event necessary to trigger the transition, the condition that must be true in order for the triggering event actually to enable the transition, and the action that results if the transition is taken (a set of generated events). A basic transition may not be taken unless it is enabled.

The definition of t11 says that it is enabled whenever TempLow_Event occurs and the condition TRUE is satisfied—i.e., whenever TempLow_Event occurs. When taken, the transition generates the event Rod1_Up. The definition of t12 says that it is enabled by the occurrence of a certain timeout when the condition TIME >= PrevTempLate() also holds; and the result is again to generate the event Rod1_Up. Transitions may generate internal

events (called "state events") that trigger other basic transitions, or "interface" events that generate external output (not represented here).

The line

```
TIME >= PrevTempLate()                    : T ;
```

is actually a degenerate representation of a table, in this case a table with one entry. A more complex table will make things clearer:

```
CONDITION :   TABLE
   A          : F T ;
   B          : T T ;
   C          : T . ;
   D          : . F ;
END TABLE
```

Each column of the table represent a single scenario defining a boolean combination of the formulas `A`, ..., `D`; and the table as a whole represents the assertion that at least one of these scenarios is true. The two scenarios are:

- `(not A) and B and C`

  The three conjuncts of this formula are determined by the table's first column: the `F` opposite `A` says that, in this scenario, `A` is false, and therefore yields the conjunct `not A`; the `T`'s opposite `B` and `C` assert that both formulas are true in this scenario, yielding the conjuncts `B` and `C`, respectively; and the `.` opposite `D` means that in this scenario the value of `D` is irrelevant.

- `A and B and (not D)`

  The conjuncts of this formula are similarly defined by the second column.

Thus a table defines a boolean combination of the predicates occurring in its leftmost column.

3.3 Transitions of the RSML machine

The semantics of an RSML machine is defined by saying how the basic transitions are combined to define the configuration changes and state changes of

the whole machine. Notice first that two basic transitions may not be compatible with one another. For example, the transition from `OkToMove2` to `Up2` is incompatible with the transition from `OkToMove2` to `Down2`: we cannot take both transitions at once, since the states `Up2` and `Down2` cannot both be part of the same configuration. On the other hand, any of the basic transitions within `Rod1` are compatible with any of the transitions within `Rod2`, since these two states represent concurrent processes. The precise definition of "compatible" raises some subtle questions, involving both mathematical problems and design decisions. These are discussed in section 9.

The global transitions of the RSML machine can be defined, somewhat informally, as follows: Some external event arrives "at the boundary" of the top-level state `Rod_Control`. This generates some collection of internal events, whose occurrence enables certain of the basic transitions. We choose, nondeterministically, a maximal set of mutually compatible enabled transitions and then take them all at once, generating a new configuration and a new set of internal events. This completes the first *microstep*. These newly generated events and the new configuration determine once again a set of enabled transitions; and, as before, we choose a maximal compatible set of those and take all of them at once, completing the next microstep. We keep on taking microsteps until no more internal events are generated; and that concludes the single step of the RSML machine caused by the arrival of the initial external event. (Execution of the microsteps may also generate externally visible events, which we are for the present ignoring.) If the sequence of microsteps does not terminate, then the RSML machine contains a semantic error.

We note, but set aside, a few subtle questions. For example, should effects generated earlier in the sequence of microsteps remain invisible until the sequence is over, or should they be visible to (and potentially affect) later microsteps? The answer is a design decision, to be based on practical considerations.

3.4 Analysis

The paper [9] estimates that the fewer than 200 states in the TCAS II specification correspond to 10^{40} configurations. Analysis of a model must therefore avoid generating its entire state space. The paper [8] describes certain forms of analysis that can be conducted piecewise, such as:

- No internal event is ignored—i.e., that every event, in every possible situation, enables at least one basic transition.

- The RSML machine is deterministic. A sufficient, but not necessary, condition is that in any situation that can arise there is only one maximal consistent set of enabled transitions—that is, all enabled transitions are compatible.

- An infinite sequence of microsteps cannot arise.

Current RSML tools apply conservative tests to check these properties, and more powerful tests based on theorem-proving could be a useful improvement. For example, to check that an internal event is not ignored it suffices to form the disjunction of all the conditions on all the transitions that the event triggers and check that this disjunction is equivalent to the formula "true". The current RSML tools do this by purely propositional reasoning, treating conditions like "`X > 0`" and "`X <= 0`" as distinct and independent propositional atoms, and ignoring the logical connections between them. As a result the tools may generate false warnings, raising the possibility of a problem when none exists. More sophisticated theorem-proving should be able to eliminate those.

Another example is the check to rule out an infinite sequence of microsteps. Current tools generate a directed graph in which the nodes are events, and an edge leads from e_1 to e_2 if and only if a transition triggered by e_1 can generate e_2 as one of its actions. A sufficient, but not necessary, condition for ruling out infinite sequences of microsteps is that this graph be acyclic. Again, we should be able to harness theorem-proving to provide a more accurate test.

4 Logical interface

By a "logical interface" to RSML we mean an encoding of the information in an RSML specification so that it can be exploited by theorem provers or other analysis tools to answer questions of interest.

We first provide two examples:

- a strategy for encoding the state hierarchy

- the table of compatible transitions

and then discuss general requirements for defining a useful logical interface.

4.1 Encoding hierarchical states

Consider the state hierarchy defined in figures 2 and 3.

Conditions on transitions may contain assertions of the form "the current configuration contains state `On`"—which we'll abbreviate for now as "`in_state(On)`." Theorem provers will be called on to reason about propositions containing instances of the `in_state` predicate. We could model `in_state` explicitly by first defining the set of all configurations; then defining `in_state` in terms of the model of configurations; and so on.

That would be a terrible decision. It is possible to model the logical relations among the `in_state` assertions about configurations without having to model the configurations themselves. We first associate each state with a boolean constant (for which we'll use the same name). Thus we declare boolean constants `Rod_Control`, `Off`, `ShutDown`, etc. Intuitively, we use the constant `Off` to represent the assertion `in_state(Off)`, etc. The relations among the `in_state` propositions are captured by axioms expressing relations among these constants. For example:

Being in the AND-state `On` means being in all three of its children. We an express this with an "and-parent" axiom

```
On -> Temp and Rod1 and Rod2
```

Being in the OR-state `Rod_Control` requires being in exactly one of its children, which is expressible by the combination of an "or-parent" axiom

```
Rod_Control -> Off or ShutDown or On
```

with "incompatible-or-siblings" axioms

```
Off -> (not ShutDown) and (not On)
ShutDown -> not On
```

The other incompatibilities are logical consequences of these two.

Being in a child state implies being in its parent state, expressible by such (child-implies-parent) axioms as

```
OkToMove1 -> Rod1
```

In this way we can capture the logic of the `in_state` predicate with a decidable theory the number of whose axioms does not grow too much faster than the number of states in the hierarchy.

There are many equivalent ways to formulate such a theory, and certain formulations may be better tuned for different theorem provers. One strategy that will in some sense always work is this: Let Φ be the conjunction of all the axioms. Then, in order to test whether the formula Ψ is true in the model, we could test the formula $\Phi \rightarrow \Psi$. This strategy has two drawbacks: The first is that it may be unacceptably inefficient, burdening the theorem prover with a large formula containing many unnecessary hypotheses. The second is that the explicit presence of Φ is a kind of noise that may make it harder to interpret the results when analysis indicates that there is a problem—for example, analysis may return a simplified formula in which components derived from Φ are entangled with those still remaining from Ψ.

The first difficulty could be partly alleviated by optimizing Ψ: We'll say that a state `foo` occurs minimally within the formula Φ if `foo` occurs in Φ but no descendants of `foo` occur in Φ. Let Ψ' be the conjunction of all axioms mentioning either minimal states occurring in Φ or ancestors of those minimal states. To test the truth of Φ in the model it suffices to test the truth of $\Psi' \rightarrow \Phi$. As a rule we expect this to be a smaller formula.

The second difficulty will in general require us to exploit details of a particular tool. Here, for example, is an intelligent encoding of the hierarchy within the EVES prover. EVES permits users to add heuristic labels to axioms, indicating how they are to be used in automated proof steps. Automatic simplifications rearrange formulas into if-then-else normal forms and traverse those forms, simplifying as the traversal proceeds. At any point in the traversal the simplifier has a context of assumptions, which includes a record of the branches taken in order to reach the present traversal point. That is, if the traversal is somewhere on the "`A`" branch of

```
if foo then A else B
```

"`foo`" belongs to the context; but if it's somewhere on the "`B`" branch the context contains "`not foo`" instead. If the axiom `X -> Y` is given the heuristic label of a *forward rule*, then whenever `X` belongs to the context, `Y` is automatically added.

To use EVES efficiently, we want to add forward rules in such a way that, as nearly as possible, all and only the rules "necessary" to the analysis will fire. (Getting only the necessary rules to fire is more or less equivalent to optimizing from Ψ to Ψ'. Keeping the effect of those rules in the background, as context, eliminates the noise.) Our experiments suggest that the following strategy works well. Add, as forward rules, all axioms in the categories

and-parent, incompatible-or-siblings, child-implies-parent. But the rules in the remaining category, or-parent, are replaced by equivalent forward rules suggested by the following example. Replace

```
parent -> child1 or child2
```

by the two axioms

```
not child1 -> parent = child2
not child2 -> parent = child1
```

As an example, section 7 provides the automatic encoding of the state hierarchy of figures 2 and 3 into an EVES theory.

4.2 The compatibility table

As already noted, defining which basic transitions are compatible with one another is a somewhat subtle matter. The definition we have implemented is described in section 9; and, if n is the number of basic transitions in the model, computing the $n \times n$ table requires an $O(n^3)$ calculation (via Warshall's transitive closure algorithm).

As part of the "core" semantic calculation we store the table in a global data structure that can be queried by any of the translations. So, for example, the translation to a SPIN model generates code initializing a constant array that represents this table. A SPIN execution must repeatedly consult this array in order to determine which transitions will be taken in each microstep.

4.3 Requirements for a logical interface

By a "logical interface to RSML" we mean a set of well-analyzed encodings, like the encoding of the state hierarchy described above, usable as a tool-kit for representing RSML semantics in a variety of formal tools. This requires both *a priori* analysis and experience with the peculiarities of several tools, so as to make the analysis generally useful. The discussion below applies to theorem-provers, as these have been the main subjects of our experiments.

We have been writing formulas in a kind of typewriter notation, using "`->`" for implication, using "`or`" as an infix operator, etc. The obvious strategy for communicating with tools using different notations is (speaking in OO jargon) to implement a fairly general formula class in which each formula has methods for writing itself in the notation of various tools. Thus,

much of the formula generation is done once and for all in the core of the FMI and much of the task of integrating a new tool with the FMI will amount to no more than implementing an additional "print" method for each formula. The PVS encoding of the state hierarchy is done in precisely this way. The EVES encoding could be done that way as well, but in order to take advantage of the "forward rule" mechanism, we give the "or-parent" axioms special treatment.

Note that we want to generate not only formulas, but entities like definitions, axioms, tactics, etc., of which formulas are only elements; and these entities may be named, may contain heuristic flags, etc., depending on the target tool. It would certainly be desirable if abstract forms of these entities could be generated once and for all by the core of the FMI.

Presently we are using S-expressions as a simple, reasonably general abstract representation of formulas. Future work on the FMI should address the problem of devising a more general abstract logical language. Some work has already been done on general logical languages. See, for example [4, 3].

5 Internal interface

This section consistently uses the following terminology: a "user" is a user of the FMI; "development" means integrating a tool into the FMI and a "developer" is someone trying to perform such an integration. As far as the phase I prototype is concerned the goal of development is a modest one: code that accepts an RSML definition in textual format and generates the preambles and scripts needed to use a formal tool for various forms of automatic analysis. Our goal in phase I is to show that EVES, PVS, and SPIN can be systematically integrated in this way.

The "internal interface" is a developer's view of the FMI. We provide a developer with two main resources: certain core functions that carry out as much of the analysis as we can in an abstract way (for example, by generating formulas that we think a theorem prover will require either as axioms or as candidate theorems); "templates" for writing any supplementary code that he needs.

5.1 Ox

Ox [1] is a simple language that allows a developer to add attribute grammar equations ([6],[7]) to a YACC description of the syntax of RSML, and also to associate actions with each production. It permits us to declare attributes

for each symbol of the grammar and define their values, inductively, in terms of the values of other attributes from the productions in which that symbol occurs. For example, the attributes for the `state_def` symbol, denoting the definition of a state, include the following:

- *code*, of type integer, that assigns a unique integer code to each state
- *ancestor*, a string, that gives the name of its parent state (if any)
- *children_imply_parent*, a pointer to an S-expression that represents the "children-imply-parent" axiom for this state

The actions of a production may refer to the attributes defined for the production and may invoke auxiliary code written in C++. (Ox will also accept auxiliary code written in C.) Actions define what to do with the information obtained from the attributes. For example, one action associated with the `state_def` production generates the EVES axiom asserting the *children_imply_parent* formula as an EVES forward rule. Another action associated with `state_def` generates a line of SPIN code which defines the state name as a synonym for its integer code (using the C-style `#define` notation).

The Ox processor produces an *evaluator*—written in Lex, Yacc, and C++—which is compilable on a C++ compiler. Execution of the compiled evaluator proceeds in two stages. The first stage builds a parse tree, finds an order in which to evaluate attributes, and computes the attributes. The second performs traversals of the parse tree (in an order specified by the developer) and executes the actions for each traversal. The actions may be written in C++ and may call on C++ auxiliary functions as well as the values of the attributes.

It is thus possible to define the attributes in a clean and simple, essentially functional, style. And the traversal mechanism simplifies the job of generating the desired formal models.

5.2 The core code

The core of the Ox code we supply comes in two parts. The first, which does not concern a developer, is a parser for the RSML language. The second is a set of attributes that define basic static semantic and logical information that developers will need to exploit. One of our principal Phase I tasks has been a first cut at deciding what that collection of information and attributes should be. In addition, we supply C++ classes implementing the

basic data types that developers will need—such as the class of S-expressions and methods for manipulating them.

A note on parsing. We assume that the input text, having been checked or generated by the RSML front-end, is legal. This permits us to build our parser in Ox by adapting, and simplifying, RSML source code. A first lexing pass generates a global data structure containing enough information to make the next pass (lexing and parsing) unambiguous. Our simplification consists primarily in gathering the absolute minimum of such information (granted the assumption that the definition is legal).

5.3 The developer's template

The developer's template is a noweb ([10]) file. The suite of noweb tools provides a simplified variant of Knuth's web programs, which permit a developer to write code fragments and documentation in a logical or expository order and then to generate either compilable code or a printable document by applying appropriate utilities. The utilities rely on a collection of "modules" and source code that labels the fragments to indicate which modules they belong to. The modules can be thought of as the headings and subheadings of an outline for the code, and the notangle utility gathers up the entries within each module to fill in the outline. The attached documentation of the prototype code was produced by processing the noweb source with notangle, configured to generate LaTeX source.

Leaving aside its virtues as a development and documentation tool, we use noweb to provide a poor man's version of a menu-driven installation program that helps systematize some of the developer's tasks. For example, the template includes the following kinds of instructions.

- In module "Globals", supply the name of the traversal, declared as an integer variable initialized to 0.

  The developer edits the given template

  ```
  <<Globals>>=
          int *FOO* = 0;
  ```

  to replace the placeholder *FOO* with the name of the desired traversal, for example

  ```
  <<Globals>>=
          int eves = 0;
  ```

- In module "Traversals", supply an Ox declaration of that variable as a traversal.

  In this case the developer integrating EVES edits the template text by supplying the phrase `preorder eves`, stipulating a preorder traversal, to produce:

```
<<Traversals>>=
traversal preorder eves
```

- Define any global state that the translation needs, putting any necessary type definitions in an "Includes" module, and the state itself in the "Globals" module.

  In this case the developer edits the template text by supplying `"eves_state.h"` and the declaration of `my_eves_state`:

```
<<Include files>>=
#include "eves_state.h"

<<Globals>>=
eves_state my_eves_state;
```

And so on. (It would of course be possible to automate this further into a true installation program, but that hardly seems worth the effort.) All the files that we ask the developer to edit are **noweb** files. We expect the developer to follow the **noweb** paradigm for filling them in. The auxiliary C++ code (such as the class `eves_state`) can be created however the developer likes—written directly on a text editor, for example, or generated by **noweb**.

The meat of the translation code, of course, is not so trivial. Our template supplies:

- for each grammar symbol, a **noweb** module in which to declare additional attributes for that symbol;
- for each production, a module in which to define the attributes so declared;
- for each production, a module in which to define the traversal actions for that production.

We have already supplied the outline that **notangle** will use to assemble these declarations and definitions into appropriate input for Ox. As noted, our goal is to supply a collection of predefined attributes so rich that little or no new work needs to be done.

5.4 The traversal code

Our prototype strategy uses the traversal code—which can, in principle, be arbitrary—to generate yet another **noweb** file. This is a convenient trick: the input to formal tools is usually organized in highly stylized ways, and we use this generated **noweb** text to provide a multiplexing filter that distributes pieces into the proper places.

The one piece of real **noweb** programming developer must do is to define a "skeleton," a top-level **noweb** outline defining the modules of which the formal model will consist and their internal organization. In the case of the EVES. for example, that skeleton provides modules for declarations of state names, for the child-implies-parent rules, for the incompatible-or-sibling rules, etc. Then, each traversal action must generate EVES text labeled with the appropriate module name, so that **notangle** will organize the output properly.

This organization of the prototype code has proven simple and effective, and permits relatively simple experimentation with variations in translation strategies.

6 Results

6.1 RSML semantics

As already noted, we have collaborated with Mats Heimdahl to provide a better semantic description of RSML, which will be included in [8]. Section 9 contains some additional work on RSML semantics and a discussion of some open problems.

6.2 Prototype code

The current prototype code parses the complete RSML language and, for specifications written in a subset of RSML (see section 6.4), defines attributes containing the following semantic information:

- S-expressions encoding the following features of the RSML model:

  - The state hierarchy (as described in section 4.1).
  - The declarations and definitions of RSML constants.
  - The declarations and definitions of RSML functions.
  - Declarations of RSML input variables.

- S-expressions encoding the following analytical assertions about the RSML model:

  - Each RSML function is well-defined—the table of cases defining it is exhaustive and exclusive.
  - Each conditional state is well-defined—the conditions defining exit transitions from the state is exhaustive and exclusive.
  - Each state event is guaranteed to enable at least one transition.

- The global table defining which pairs of basic transitions are compatible (see section 4.2).

Note: in the logical models (such as EVES and PVS), the basic transitions are currently represented by the way in which they contribute to the analytical assertions; whereas the SPIN model contains an explicit operational representation of each basic transition.

The prototype defines traversals for EVES, PVS, and SPIN. The EVES traversal generates an EVES version of the theory described above and in section 4.1, and also generates as candidate EVES theorems the list describe above of analytical assertions about the specification. The PVS traversal does some but not all of the corresponding things in PVS terms—it was undertaken as a brief experiment to see how rapidly it could be done, and little effort would be needed to complete it. The SPIN traversal generates an executable SPIN model of the specification. (For want of time, this model dummies out all the conditions governing transitions to the condition `TRUE`.)

6.3 Systematic integration of tools

Our noweb templates have proven a convenient way to integrate tools to the RSML front end. The initial set of attributes and actions was defined solely with EVES in mind, but served quite well to permit a rapid addition of both PVS and SPIN.

We also implemented a number of purely internal tools for generating large pieces of the necessary templates automatically from the RSML grammar. (These tools guarantee the internal consistency of the templates, and will make it simple to adapt to changes in RSML.)

Finally, the noweb source makes all the code and its documentation easily reviewable.

6.4 The RSML fragment

Here is an informal description of the restrictions our prototype code currently places on RSML specifications (roughly classified according to their underlying reasons).

To simplify certain parsing problems we currently assume

- All names occurring in the RSML text, other than local variables, are distinct.
- At most 1,000 names occur in the specification.
- A system has only one component.

Because of unresolved semantic questions we do not currently provide (full) support for the following language features:

- arrays of OR-states, arrays of AND-states, and transition busses. Applying the translators to code that contains these may result in a crash, or may generate a meaningless model.
- Conditional states. The SPIN code output by a translator encountering conditional states will not, in general, compile.
- Real literals. The SPIN translation treats them like integers, and assumes that they have the form [[xxx.0]].
- Timeout events and conditions referring to time.
- The expected range and granularity of variables.

For lack of time, we have omitted RSML macros and have used a simplified treatment of RSML variables. We have omitted output variables and modeled input variables simply as containers for values—a variable should actually be modeled as a function mapping times to values (i.e., as a collection of time-value pairs).

6.5 Applications

We have spent limited time in experimenting with the results. Some observations:

EVES and PVS easily handle the model of hierarchical states—i.e., easily decide propositions making assertions about it. The EVES prover automatically invokes the necessary rules, but to use PVS we must generate a PVS tactic that invokes the hierarchy axioms as lemmas. (In experiments, that tactic has been written by hand.)

Reasoning automatically about real number properties presents some difficulties in EVES, as the EVES library does not contain a real-number theory. In addition, the EVES heuristics have certain intrinsic weaknesses for dealing with such things as transitive relations (like $\leq$ on the reals). Technically, the limitation is that conditional rules may have only one premise. PVS is more immediately helpful, as its built-in simplification strategies include basic real number arithmetic. As noted in section 9.1, however, the proper RSML semantics for real numbers is unclear; and without a firm decision on the semantic model it seemed unwise to devote too much time to worrying about theorem-proving support.

SPIN provides two modes: interactive simulation through a GUI, and "validation" (i.e., automated model-checking). Brief experiments suggest that it is impractical to apply interactive simulation to the models we generate. The problem, we conjecture, is the communications overhead between the executing model and the GUI. Runs in the "validation" mode, for a prescribed sequence of concrete inputs, were much more efficient. However, we currently generate code that is purely deterministic: In each microstep the search for a maximal set of compatible enabled transitions proceeds in a fixed order. A truly non-deterministic search might be much more costly, though we have not experimented with that. By slightly modifying the generated code, and at virtually no cost in efficiency, the model could choose the transitions deterministically but raise a warning if a different choice would be possible. We could also provide a flag that would permit a choice between deterministic and non-deterministic models. An interesting application of SPIN that we have not been able to investigate is as follows: SPIN can check temporal logic assertions about the execution of the model, where the assertions may refer to named control points within the model's code. We could generate a set of predefined names for predefined control points and thereby provide the user with a useful language for making checkable assertions about a specification.

7 The EVES translation

Here is the EVES encoding of the state hierarchy defined in figures 2 and 3.

```
(function-stub Rod_Control ())
(function-stub Off ())
(function-stub ShutDown ())
(function-stub On ())
(function-stub Temp ())
(function-stub HighTemp ())
(function-stub LowTemp ())
(function-stub Rod1 ())
(function-stub JustMoved1 ())
(function-stub Up1 ())
(function-stub Down1 ())
(function-stub OkToMove1 ())
(function-stub Rod2 ())
(function-stub JustMoved2 ())
(function-stub Up2 ())
(function-stub Down2 ())
(function-stub OkToMove2 ())
(function-stub Desired_Temp ())
(function-stub T_Min ())
(function-stub T_Max ())
(function-stub Temp_Gran ())
(function-stub Wait ())
(function-stub Temp_Reading_Timeout ())
(function-stub Temp_Reading_Timeout_2 ())
(function-stub Temp_Reading_Timeout_3 ())
(function-stub Temp_Separation ())
(function-stub Up ())
(function-stub Down ())
(function-stub None ())
(function-stub PrevTempLate ())
(function-stub PrevTempHigh ())
(function-stub PrevTempLow ())
(function-stub TempReportTimeoutLimit ())
(function-stub Rod2_Cannot_Move_Time ())
(function-stub Temperature ())
(function-stub Temp_Status ())
(function-stub Temp_Out ())
(function-stub Rod_1_Setting ())
(function-stub Rod_2_Setting ())
(frule child-implies-parent__10 () (IMPLIES (On) (Rod_Control)))
```

```
(frule child-implies-parent__11 () (IMPLIES (ShutDown) (Rod_Control)))
(frule child-implies-parent__12 () (IMPLIES (Off) (Rod_Control)))
(frule child-implies-parent__15 () (IMPLIES (Rod2) (On)))
(frule child-implies-parent__16 () (IMPLIES (Rod1) (On)))
(frule child-implies-parent__17 () (IMPLIES (Temp) (On)))
(frule child-implies-parent__19 () (IMPLIES (LowTemp) (Temp)))
(frule child-implies-parent__20 () (IMPLIES (HighTemp) (Temp)))
(frule child-implies-parent__22 () (IMPLIES (OkToMove1) (Rod1)))
(frule child-implies-parent__23 () (IMPLIES (JustMoved1) (Rod1)))
(frule child-implies-parent__25 () (IMPLIES (Down1) (JustMoved1)))
(frule child-implies-parent__26 () (IMPLIES (Up1) (JustMoved1)))
(frule child-implies-parent__28 () (IMPLIES (OkToMove2) (Rod2)))
(frule child-implies-parent__29 () (IMPLIES (JustMoved2) (Rod2)))
(frule child-implies-parent__31 () (IMPLIES (Down2) (JustMoved2)))
(frule child-implies-parent__32 () (IMPLIES (Up2) (JustMoved2)))
(frule parent-implies-children__18 ()
  (IMPLIES (On) (AND (Rod2) (Rod1) (Temp)))
)
(frule incompatible-or-sibs__13 ()
  (IMPLIES (On) (AND (NOT (ShutDown)) (NOT (Off))))
)
(frule incompatible-or-sibs__14 () (IMPLIES (ShutDown) (AND (NOT (Off)))))
(frule incompatible-or-sibs__21 () (IMPLIES (LowTemp) (AND (NOT (HighTemp)))))
(frule incompatible-or-sibs__24 ()
  (IMPLIES (OkToMove1) (AND (NOT (JustMoved1))))
)
(frule incompatible-or-sibs__27 () (IMPLIES (Down1) (AND (NOT (Up1)))))
(frule incompatible-or-sibs__30 ()
  (IMPLIES (OkToMove2) (AND (NOT (JustMoved2))))
)
(frule incompatible-or-sibs__33 () (IMPLIES (Down2) (AND (NOT (Up2)))))
(function-stub Rod_Control~1 ())
(frule not-On->Rod_Control=Rod_Control~1 ()
  (IMPLIES (NOT (On)) (= (Rod_Control) (Rod_Control~1)))
)
(frule not-Rod_Control~1->Rod_Control=On ()
  (IMPLIES (NOT (Rod_Control~1)) (= (Rod_Control) (On)))
)
(frule not-ShutDown->Rod_Control~1=Off ()
  (IMPLIES (NOT (ShutDown)) (= (Rod_Control~1) (Off)))
)
(frule not-Off->Rod_Control~1=ShutDown ()
  (IMPLIES (NOT (Off)) (= (Rod_Control~1) (ShutDown)))
)
```

```
(frule not-LowTemp->Temp=HighTemp ()
  (IMPLIES (NOT (LowTemp)) (= (Temp) (HighTemp)))
)
(frule not-HighTemp->Temp=LowTemp ()
  (IMPLIES (NOT (HighTemp)) (= (Temp) (LowTemp)))
)
(frule not-OkToMove1->Rod1=JustMoved1 ()
  (IMPLIES (NOT (OkToMove1)) (= (Rod1) (JustMoved1)))
)
(frule not-JustMoved1->Rod1=OkToMove1 ()
  (IMPLIES (NOT (JustMoved1)) (= (Rod1) (OkToMove1)))
)
(frule not-Down1->JustMoved1=Up1 ()
  (IMPLIES (NOT (Down1)) (= (JustMoved1) (Up1)))
)
(frule not-Up1->JustMoved1=Down1 ()
  (IMPLIES (NOT (Up1)) (= (JustMoved1) (Down1)))
)
(frule not-OkToMove2->Rod2=JustMoved2 ()
  (IMPLIES (NOT (OkToMove2)) (= (Rod2) (JustMoved2)))
)
(frule not-JustMoved2->Rod2=OkToMove2 ()
  (IMPLIES (NOT (JustMoved2)) (= (Rod2) (OkToMove2)))
)
(frule not-Down2->JustMoved2=Up2 ()
  (IMPLIES (NOT (Down2)) (= (JustMoved2) (Up2)))
)
(frule not-Up2->JustMoved2=Down2 ()
  (IMPLIES (NOT (Up2)) (= (JustMoved2) (Down2)))
)
(rule top-level-state__1 () (= (Rod_Control) (true)))
```

8 A SPIN model

This section explains the SPIN model by an example that annotates representative fragments of an RSML specification with the corresponding Promela code. The only alterations we make in the generated code (aside from omissions) are slight reformattings. The Promela code is completely intelligible to a human reader (thanks to careful use of the SPIN preprocessor).

The Promela code begins with a trivial preamble defining `TRUE` to be 1 and `FALSE` to be 0.

We assume that there is only one component to the system, in this case

declared by

```
COMPONENT Reactor_One
```

The component name will be the basis for mnemonic names denoting, the main Promela process, the channel on which inputs are received, etc.

8.1 The state hierarchy

The RSML state hierarchy is defined by:

```
OR_STATE Rod_Control DEFAULT Off :
    ATOMIC Off :
    ATOMIC ShutDown :
    AND_STATE On :
        OR_STATE Temp DEFAULT HighTemp:
            ATOMIC HighTemp :
            ATOMIC LowTemp :
        END OR_STATE
        OR_STATE Rod1 DEFAULT JustMoved1 :
            OR_STATE JustMoved1 DEFAULT Up1 :
                ATOMIC Up1 :
                ATOMIC Down1 :
            END OR_STATE
            ATOMIC OkToMove1 :
        END OR_STATE
        OR_STATE Rod2 DEFAULT OkToMove2 :
            OR_STATE JustMoved2 DEFAULT Up2 :
                ATOMIC Up2 :
                ATOMIC Down2 :
            END OR_STATE
            ATOMIC OkToMove2 :
        END OR_STATE
    END AND_STATE
END OR_STATE
```

This differs from the example of figures 3 and 2 only in eliminating the conditional state, which the SPIN translation does not currently support.

In Promela we do the following things:

- Declare a boolean array `state` of length `state_count` (the number of states), which represents the global configuration of states.
- Define state names as indexes into that array.

- Recursively define, for each state foo, the following actions:
  - set_parent_foo, which sets state[foo] and all its ancestors to TRUE.
  - leave_foo, which sets state[foo] and all its children to FALSE
  - enter_foo, which sets to TRUE: state[foo], all its ancestors, and the "default" children of foo

```
#define state_count 17
bool state[state_count];


#define Off 0
#define ShutDown 1
#define HighTemp 2
...

#define set_parent_Rod_Control state[Rod_Control]=TRUE
#define set_parent_Off state[Rod_Control]=TRUE; set_parent_Rod_Control
#define set_parent_ShutDown state[Rod_Control]=TRUE; set_parent_Rod_Control
#define set_parent_On state[Rod_Control]=TRUE; set_parent_Rod_Control
#define set_parent_Temp state[On]=TRUE; set_parent_On
#define set_parent_HighTemp state[Temp]=TRUE; set_parent_Temp
...

#define leave_Off state[Off]=FALSE
#define leave_ShutDown state[ShutDown]=FALSE
#define leave_HighTemp state[HighTemp]=FALSE
#define leave_LowTemp state[LowTemp]=FALSE
#define leave_Temp state[Temp]=FALSE; leave_LowTemp; leave_HighTemp
...
#define leave_Rod_Control state[Rod_Control]=FALSE; leave_On; \
                          leave_ShutDown; leave_Off

#define enter_Off set_parent_Off; state[Off]=TRUE
#define enter_ShutDown set_parent_ShutDown; state[ShutDown]=TRUE
#define enter_HighTemp set_parent_HighTemp; state[HighTemp]=TRUE
#define enter_LowTemp set_parent_LowTemp; state[LowTemp]=TRUE
#define enter_Temp set_parent_Temp; state[Temp]=TRUE; enter_HighTemp
#define enter_Up1 set_parent_Up1; state[Up1]=TRUE
...
#define enter_Rod_Control set_parent_Rod_Control; \
                          state[Rod_Control]=TRUE; enter_Off
```

8.2 Constant declarations

We make the simplifying assumption that all constants are of type integer, thus an RSML declaration such as

```
CONSTANT Desired_Temp :
        VALUE : 400.0
END CONSTANT
```

becomes the following trivial Promela code:

```
#define Desired_Temp 400
```

8.3 Events, input variables, and input interfaces

The event, in variable, and in interface declarations such as

```
EVENT Power_On STATE;
EVENT Power_Off STATE;
...
EVENT Rod1_Down INTERFACE;
EVENT Rod1_Up INTERFACE;
...
EVENT Set_Rod1 VARIABLE;
EVENT Set_Rod2 VARIABLE;

IN_VARIABLE Temperature :
        TYPE : NUMERIC
        EXPECTED_MIN : 0.0
        EXPECTED_MAX : 100.0
        MIN_GRAN : 1.0
        MAX_GRAN : 1.0
END IN_VARIABLE

IN_VARIABLE Temp_Status :
        TYPE : NUMERIC
        EXPECTED_MIN : 0.0
        EXPECTED_MAX : 100.0
        MIN_GRAN : 1.0
        MAX_GRAN : 1.0
END IN_VARIABLE


IN_INTERFACE OnButton :
```

```
        SOURCE   : EXTERNAL
        TRIGGER  : RECEIVE ()
        SELECTION : TRUE
        ACTION    : Power_On
END IN_INTERFACE

IN_INTERFACE OffButton :
        SOURCE   : EXTERNAL
        TRIGGER  : RECEIVE ()
        SELECTION : TRUE
        ACTION    : Power_Off
END IN_INTERFACE

IN_INTERFACE TempSensor :
        SOURCE   : EXTERNAL
        TRIGGER  : RECEIVE (Temp_Status, Temperature)
        SELECTION : TRUE
        ACTION    : Temp_Report_Event
END IN_INTERFACE

IN_INTERFACE Testing :
        SOURCE   : Reactor_One
        TRIGGER  : RECEIVE (Temperature)
        SELECTION : TRUE
        ACTION    : Temp_Report_Event
END IN_INTERFACE
```

are modeled by Promela code which does the following:

- Declare the type of the inputs to be received and a channel for receiving them. The name of the channel is derived from the name of the component. The input type has a boolean component for each interface (representing receipt of an input at an interface) and a byte-sized component, representing the data received.
- Declare a boolean array, `event`, of length `event_count` (the number of events), representing the set of currently "active" events.
- Define event names as indices into that array.

```
typedef In_Interface_Reactor_One{
        byte Temperature;
```

```
        byte Temp_Status;
        bool OnButton;
        bool OffButton;
        bool TempSensor;
        bool Testing;
};

chan In_Channel_Reactor_One = [1] of {In_Interface_Reactor_One};

#define event_count 12
bool event[event_count];

#define Power_On 0
#define Power_Off 1
#define ShutDown_Event 2
...
```

Input variables are represented as follows: The main Promela process will declare a local variable, `input`, of type `In_Interface_Reactor_One`, which receives the values transmitted through the interface. Thus `input.Temperature`, for example, could be used to model the in variable `Temperature`. However, we find it convenient to be sightly more roundabout: We will model the in variable `Temperature`, for example, by declaring a local variable `Temperature` of the main process and copying the value of `input.Temperature` into it.

8.4 Output events, variables, and interfaces

Output events, output variables, and interfaces are currently omitted.

8.5 Functions and macros

Functions and macros are currently omitted.

8.6 Transitions

Here are two sample transition definitions:

```
TRANSITION t2 FROM Off TO On :
        TRIGGER   : Power_On
        CONDITION : TRUE
        ACTION    :
END TRANSITION
```

```
TRANSITION t3 FROM On TO ShutDown :
        TRIGGER   : Temp_Report_Event
        CONDITION :          TABLE
                Temperature > T_Max              : T . . . ;
                Temperature < T_Min              : . T . . ;
                Temperature != PrevTempHigh()    : . . T . ;
                Temperature != PrevTempLow()     : . . T . ;
        END TABLE
        ACTION    : ShutDown_Event
END TRANSITION
```

In the Promlea code we do the following things:

- Declare a boolean array, `trans_enabled`, of length `transition_count` (the number of transitions), representing the set of currently enabled transitions; and a similar array, `selected` representing the set of currently selected transitions.
- Declare boolean array, `compat`, representing the (constant) table specifying which pairs of transitions are compatible. (As Promela supports only one-dimensional arrays it's convenient to introduce a macro, `compatible(i,j)`, defining the indexing scheme whereby this array can be thought of as two-dimensional.)
- Define for each transition `Foo`
  - `cond_Foo`, representing the enabling condition for the transition. (This is currently dummied out to `TRUE`. Note that RSML permits conditions that are not executable.)
  - `enabled_Foo` defining what it means for `Foo` to be enabled: the current configuration must contain the state at the tail of the arrow, the triggering event must be currently active, and the condition must be true;
  - `take_Foo` defining the effect of taking transition `Foo`: we exit its "leaving" state enter its target state, and "consume" the triggering event; we also deselect `Foo`;
  - `action_Foo` defining the action taken: the events generated by the transition become active.

```
#define transition_count 16
```

```
bool trans_enabled[transition_count];
bool selected[transition_count];
#define t_count_squared 256
bool compat[t_count_squared];
#define compatible(i,j) compat[i * transition_count + j]

...

#define t2 1
#define cond_t2 TRUE
#define enabled_t2 state[Off] && event[Temp_Report_Event] && cond_t2
#define take_t2 selected[t2]=FALSE; leave_Off; enter_On; \
                event[Temp_Report_Event]=FALSE
#define action_t2

#define t3 2
#define cond_t3 TRUE
#define enabled_t3 state[On] && event[Temp_Report_Event] && cond_t3
#define take_t3 selected[t3]=FALSE; leave_On; enter_ShutDown; \
                event[Temp_Report_Event]=FALSE
#define action_t3 event[ShutDown_Event]=TRUE

...
```

8.7 The main Promela process

So far we have generated nothing but declarations and definitions. The Promela process representing Reactor_One is an infinite loop that accepts an input from its input channel, executes the resulting sequence of microsteps, and repeats. The outline is:

```
proctype Reactor_One() {

        /* Declare local variables */

        /* Initialize arrays */

        /* Enter the top-level state */

        enter_Rod_Control;

        /* Outer loop */

end: do /* Top of outer loop */
```

```
:: In_Channel_Reactor_One ? input;

atomic{

do  /* Top of microstep loop */

        /* Perform sequence of microsteps */

od; /* End of micro step loop */

}

        /* Perform output actions */

od /* End of outer loop */
}
```

The label "end:" prefacing the initial "do" tells SPIN not to report an error when the main process blocks at this point because the driver process has stopped sending input through In_Channel_Reactor_One.

Placing the code for a step inside "atomic" command instructs SPIN to treat it as an atomic step.

8.7.1 Declare local variables

The local variables consist of a loop-counter, a boolean continue_microloop that is used to determine when the sequence of microsteps is complete, a variable input to receive external inputs, and models of the RSML in variables into which values of input will be copied.

```
        int i;          /* All-purpose loop counter */
        bool continue_microloop;

 /* Local variable that receives input */

        In_Interface_Reactor_One input;

 /* Models for RSML in variables */

        int Temperature;
        int Temp_Status;
```

8.7.2 Initialize arrays

The components `event`, `selected`, and `trans_enabled` must be set to `FALSE` initially and at the top of each *step*. The initial values are taken care of automatically, because all Promela variables are initialized to 0.

We must explicitly initialize the constant array `compat`, but need only bother to set its non-0 values:

```
d_step{
        compatible(4,8) = TRUE;
        compatible(4,9) = TRUE;
        compatible(4,10) = TRUE;
...
        compatible(15,10) = TRUE;
        compatible(15,11) = TRUE;
        compatible(15,12) = TRUE;
}
```

Bracketing these statements with the `d_step` indication tells SPIN to treat this as a deterministic atomic step.

8.7.3 Perform sequence of microsteps

Each microstep consists of the following steps:

- Determine the effect of the inputs (setting `event` and the values of the input variables).
- Determine which transitions are enabled (setting `trans_enabled`).
- Choose selected transitions (setting `selected` to a maximal consistent subset of the enabled transitions).
- Take the selected transitions.
- Reset arrays.
- Decide whether the microstep is finished.

Effect of the inputs The boolean components of `input` indicate which interface received a message. (According to RSML semantics we can assume that precisely one of these is set.) The effect of an input to activate certain events (setting components of `event` to true) and to set the values of the local variables representing RSML input variables.

```
if

:: input.OnButton  && TRUE -> event[Power_On]=TRUE;

:: input.OffButton  && TRUE -> event[Power_Off]=TRUE;

:: input.TempSensor  && TRUE -> event[Temp_Report_Event]=TRUE;
                                Temp_Status = input.Temp_Status;

:: input.Testing  && TRUE -> event[Temp_Report_Event]=TRUE;
                             Temperature = input.Temperature;

fi;
```

Enabling transitions

```
trans_enabled[t1] = enabled_t1;
trans_enabled[t2] = enabled_t2;
...
```

Choose selected transitions The code for doing selecting transition is straightforward: nested loops that ask, of each transition in turn, whether it is compatible with all the transitions selected so far. The order in which transitions are considered is deterministic, for efficiency's sake, but could be made non-deterministic:

```
i = 0;
do
:: i < transition_count ->

        if
        :: trans_enabled[i] && ! selected[i] ->
            int j = 0;
            selected[i] = TRUE;
            do
                ::
                if
                :: j < transition_count ->
                    if
                    :: selected[j] && ! compatible(i,j) ->
                        selected[i] = FALSE; break
                    :: else -> skip
                    fi;
```

```
                :: else -> skip
                fi;
                j++;
            od;

        :: else -> skip
        fi;
        i++;

:: else -> break
od;
```

Take the selected transitions We can take the transitions, sequentially, in any order, so we choose an order deterministically:

```
if
:: selected[t1] -> take_t1; action_t1
:: else -> skip
fi;
if
:: selected[t2] -> take_t2; action_t2
:: else -> skip
fi;
...
```

Reset arrays We must begin each microstep with the arrays `trans_enabled` and `selected` set to `FALSE`. All selected transitions are taken, and taking them deselects them, so `selected` is automatically reset.

We must reset `trans_enabled` explicitly:

```
i = 0;
do
        :: i < transition_count -> trans_enabled[i]=FALSE; i++
        :: else -> break;
od;
```

Acting on an event (e.g., taking a transition) also "consumes" its triggering event (setting the appropriate component of `event` to `FALSE`) and it is a semantic error for any events to be "unconsumed" at the end of a microstep. We do not currently test for this error.

Quit the sequence of microsteps? The microstep sequence is done if and only if no new *state* events are active.

```
continue_microloop =
event[Power_On] ||
event[Power_Off] ||
event[ShutDown_Event] ||
event[Temp_Report_Event] ||
event[TempHigh_Event] ||
event[TempLow_Event] ||
FALSE;

/* The break in this next statement quits the microloop*/

if
:: continue_microloop = FALSE -> break
:: else -> skip
fi;
```

The final "FALSE" in the assignment to continue_microloop is a convenient way to handle the pathological case in which there are no state events.

8.7.4 Perform output actions

Currently, we do not model any output actions. As with the "take" actions, these should consume the events that trigger them. Therefore, in the absence of semantic errors, all components of event will be set to FALSE at the end of this and the next step will begin with all of the arrays event, selected, and trans_enabled properly initialized.

8.8 The driver and the initial process

We run the main process in parallel with a driver that supplies inputs. The user supplies a body for the driver:

```
 proctype Driver () {
/* User-supplied body */
}

init { run Reactor_One() ;
       run Driver() }
```

9 Semantic questions in RSML

9.1 Real numbers

RSML input variables range over real numbers—sensor values, for example—and RSML functions are defined using basic arithmetical operations over them. But how should we model the range of these variables and the semantics of the arithmetical operations? There is no formal problem in stipulating that the model is simply the mathematical real numbers and real number operations; but, if so, we have a problem relating those specifications to the behavior of computations intended to represent them.

If variables are modeled as the mathematical real numbers, then the two conditions $X = Y$, $X/2 \neq Y/2$ form an exhaustive and exclusive pair, as a theorem-prover would testify. But it does not follow that precisely one of the two corresponding executable comparison operations will always yield TRUE. Depending on the precise implementation details (not only the details of the source code but those of the underlying mathematical hardware) it is quite possible to have situations in which both yield FALSE. In this case, it's not clear whether the mathematical analysis is of much use.

One possible response is to model variables with the mathematical reals, and to implement within the RSML tools automatic code-generation that implements the conditions as executable expressions guaranteed to satisfy a property such as the following: Evaluation of at least one condition will always yield TRUE; and more than TRUE result will only arise in certain well-defined boundary states. It would then be up to users to establish that, in those boundary states, the response dictated by taking any of the TRUE branches would be acceptable.

Other choices are possible, and the issue is at least as much a design problem as it is a mathematical one.

9.2 Basic transitions

So far we have informally said that an arrow defines a basic transition, and that a basic transition maps configurations to configurations.[1] Both of these informal understandings need further explanation. We assume that the reader is familiar with the terminology of the paper [8], which is included with this report as an appendix—in particular the notion of parallel states

[1] It's irrelevant to the present discussion that transitions may also generate events.

and of a consistent set of states. Following that paper we use $x \perp y$ to mean that x and y are parallel states. In what follows

- an element of *StateSet* is any set of states;
- an element of *Config* is any maximal consistent sets of states.

9.2.1 Legal arrows

We first note that not all arrows represent intuitively sensible transitions. Consider the state hierarchy defined in figures 2 and 3. An arrow from one state to a parallel sibling, such as from `OkToMove1` to `OkToMove2`, certainly seems intuitively puzzling; as does an arrow that leaps to one of a set of parallel siblings, such as an arrow from `Off` to `Rod2`. Such leaps are analogous to `goto` statements that do not respect the block structure of a language. Assigning a meaning to these leaps by brute force is possible, but seems contrary to the intention of a specification language, whose whole point is to be as clear as possible.

We therefore propose some "legality" rules that place reasonable, statically determinable, limits on the the kinds of arrows permitted. Further terminology:

- A *component* is a child of an AND-state.
- $comp(x) = \min\{z | x \leq z \text{ and } z \text{ is a component}\}$

Note that, by the definition of min, if there is no component $z \geq x$, $comp(x)$ returns the top-level state (in our example, the state `Rod_Control`). We will be applying $comp(\ )$only to OR-states, so unless x is the top-level state, $comp(x) > x$.

The first legality rule forbids jumping into a parallel components from outside. Formally,

Legality rule 1:

An arrow from x to y is illegal unless $x \leq comp(y)$.

It also seems somewhat puzzling to allow an arrow whose source or target is a component. Hence

Legality rule 2:

An arrow from x to y is illegal if x or y is a component.

From a formal point of view, rule 2 seems less important than rule 1.

We will say that an arrow from x to y is *legal* if x and y satisfy rules 1 and 2.

9.2.2 The map defined by an arrow

We'll use $x \rightarrow y$ to denote an arrow from x to y, and $A \xrightarrow{p} B$ to denote the set of partial mappings from A to B. We next associate each arrow $x \rightarrow y$ with a mapping

$$[x, y] : \mathit{Config} \xrightarrow{p} \mathit{StateSet}$$

and which, if x and y are properly restricted, guarantees that the output values of $[x, y]$ are really elements of *Config*, and not just of *StateSet*.

Intuitively, $[x, y](A)$ should be defined when $x \in A$, and it represents the act of "leaving" x and "entering" y. As a first, approximation, then, when defined, the value of $[x, y](A)$ should be obtainable by deleting x and any of its children from A, and adding y and some appropriate set of *its* children to the result (or something close to that). This follows the prescriptions of [8]. Things are not quite that simple, as can be seen by considering some examples: `Down1` $\rightarrow$ `OkToMove1` must exit not only `Down1` but also its parent `JustMoved1`. Similarly, the arrow `OkToMove1` $\rightarrow$ `Down1` must enter not only `Down1` but also `JustMoved1`.

Note that the presence of conditional states complicates this picture, since it means we cannot fix a single set of states that a transition to a condition state actually enters. For now conditional states are ignored and further discussion is deferred to section 9.4.

We list some technical definitions and lemmas (and will indicate which there has been time to prove, and which currently remain plausible conjectures):

- $up(x) = \{z | z \geq x\}$
- $down(x) = \{z | z \leq x\}$
- If x is not a component, $default(x)$ is that subset of $down(x)$ that is chosen by default. (This is easily formalized.)

We will stipulate that $[x, y]$ enters the states in $up(y) \cup default(y)$. This takes care of the difficulties noted in example `OkToMove1` $\rightarrow$ `Down1`. Stipulating that $[x, y]$ leaves the states in $down(x)$ is not good enough—e.g., it doesn't deal with the example `Down1` $\rightarrow$ `OkToMove1`—because leaving x for

y may imply leaving other states as well. Hence the next, quite technical, definition of $leaving(x, y)$:

- If $x \leq y$, $leaving(x, y) = y$
- Otherwise, $leaving(x, y)$ is the unique x' such that

$$x \leq x' \text{ and } x' \text{ is a child of } \max(x,y)$$

Finally, we can make the desired definitions, of the source and destination sets of an arrow $x \rightarrow y$

- $Src(x, y) = down(leavingxy)$
- $Dest(x, y) = up(y) \cup default(y)$

Accordingly, $[x, y]$: $StateSet \xrightarrow{p} StateSet$ is the partial map defined as follows:

- The domain of $[x, y]$ is $\{S \in Config | x \in S\}$
- The value of $[x, y](S)$ is

$$(S - (Src(x, y) \cup down(y))) \cup Dest(x, y)$$

Here is the main technical result, which applies to arbitrary x and y (not just to x and y such that $x \rightarrow y$ is legal); and the proof, which we omit, is surprisingly annoying.

Lemma 1 *If S is consistent, $x \in S$, and not $x \perp y$, then $[x, y](S)$ is consistent.*

Lemma 2 *If $x \rightarrow y$ satisfies legality rule 1, then $x \perp y$.*

The lemma does not claim that $[x, y](S)$ is an element of *Config*—i.e., that it is a *maximal* consistent set. From the formal point of view that means that there is a hole in this development. Here are some conjectures on how the picture might be completed.

Define "strongly nonparallel" as follows:

- $x \dashv y$ means that for all $y \leq x$, not $y \perp x$

This is not a symmetric notion, though obviously $x \dashv y$ implies not $x \perp y$. We can form a half-hearted conjecture:

Conjecture 3 *If $x \dashv y$ then the output values of $[x, y]$ are elements of Config.*

This conjecture gains some interest from the trivial fact that:

Lemma 4 *If $x \rightarrow y$ satisfies legality rule 1, then $x \dashv y$.*

9.3 Compatibility of basic transitions

Section 9.2 makes a reasonable beginning at defining the semantics of basic transitions. The paper [8] defines the compatibility of two basic transitions (and, by an easy extension, of any set of them) as follows: t_1 and t_2 are compatible iff they are can be composed in either order, and that composition yields the same result in either order. In what follows we will call this notion H-compatibility. If some set of transitions is H-compatible it is easy to explain the semantics of "taking them all simultaneously"—we simply apply them sequentially, in any order.

This is a reasonable definition with some possibly counterintuitive consequences. We use illustrations, once again, from figure 2. Regarded as maps, the two arrows `OkToMove1` $\rightarrow$ `Up1` are identical—but they are H-incompatible. They cannot be sequentially composed because the range of each is disjoint from the domain of the other. For the same reason, this semantics does not support one of the common explanations of a hierarchical transition—namely, the claim that arrows entering or leaving non-atomic states are merely shorthands for sets of arrows between atomic states. Suppose we tried to define arrow `Off` $\rightarrow$ `On` as a shorthand for simultaneously taking the set of atomic transitions consisting of `Off` $\rightarrow$ `Up2`, `Off` $\rightarrow$ `Up1`, and (the conditional state `C` gives us a problem) either `Off` $\rightarrow$ `HighTemp` or `Off` $\rightarrow$ `LowTemp`. This fails because these arrows are H-incompatible—once again, they can't be sequentially composed. The semantics of H-compatibility means that high-level arrows (entering or leaving non-atomic states) are emphatically not reducible to atomic arrows. That does not necessarily mean that something is "wrong." It is simply a methodological choice.

We adapt slightly the methods of [8] by noting that they permit us to adopt any definition of compatibility that is at least as strong as H-compatibility. So, we define C to be a *compatibility* if it is a set of sets of transitions with the following properties

- $T_1 \in \mathcal{C}$ and $T_2 \subseteq T_1$ implies $T_2 \in \mathcal{C}$.
- Every element of $\mathcal{C}$ is an H-compatible set of transitions.

Which compatibility to choose is a methodological matter. For methodological and mathematical reasons, and for efficiency's sake, we propose the following definitions. First, define a notion of parallelism between pairs of basic transitions:

$$(x_1 \rightarrow y_1) \perp (x_2 \rightarrow y_2) \text{ iff } x_1 \perp x_2 \text{ and } y_1 \perp y_2$$

We define $\hat{\mathcal{C}}$ to be the set of all sets T of legal basic transitions such that for all $t_1, t_2 \in T$, $t_1 \perp t_2$. We have not had time to prove the following lemma, which seems clear:

Conjecture 5 *$\hat{\mathcal{C}}$ is a compatibility.*

Using $\hat{\mathcal{C}}$ as the underlying notion of compatibility seems to make simulation about as efficient as it could hope to get, since it reduces the test for compatibility to a pairwise test. This is the notion of compatibility that we are implementing in the SPIN translation.

9.4 Conditional states

The intuitive semantics of a conditional state is that it is never part of the configuration—nominally entering a conditional state requires immediately taking a transition that leaves it. If no exiting transition is enabled, the specification contains a semantic error.

Some complexities are immediately apparent. First is the possibility of a loop among the exit transitions of conditional states. (Speaking operationally, that would mean that an individual microstep—as opposed to a sequence of microsteps—would fail to terminate.) Second, if we are following a (non-looping) sequence of conditional exits, how do we define the states in which the conditions on the exit transitions are, successively, evaluated? Without conditional states the situation is clear: The enabled transitions are determined solely by the state at the beginning of the microstep. If one of these transitions leads to a conditional state and involves further conditional transitions, it seems that the intermediate states in this sequence would become visible and would be the states in which the exit conditions of successive conditional transitions are to be evaluated. This has at least the potential for increasing both the conceptual and computational complexity

of defining a compatible set of transitions and its result. Presumably, we would need some set of restrictions on the use of conditional transitions to keep these costs within reason.

References

[1] Kurt M. Bischoff. Design, implementation, use, and evaluation of Ox: an attribute grammar compiling system based on Yacc, Lex, and C. Technical Report TR92-30, Computer Science Dept., Iowa State University, Ames, Iowa, December 1993.

[2] D. Craigen, S.G. Gerhart, and T.J. Ralston. An international survey of industrial applications of formal methods. Technical Report NIST GCR 93/626, NIST, 1993.

[3] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[4] J. D. Guttman. A proposed interface logic for verification environments. Technical Report M91-19, The MITRE Corporation, 1991.

[5] D. Harel. Statecharts: a visual formalism for complex systems. In *The Science of Computer Programming, volume 8*, pages 231–274, 1987.

[6] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, June 1968.

[7] D.E. Knuth. Semantics of context-free languages: Correction. *Math. Syst. Theory*, 5(1):95–96, March 1971.

[8] Nancy G. Leveson and Mats Per Erik Heimdahl. Completeness and consistency in hierarchical state-based requirements. To appear in IEEE Transactions on Software Engineering.

[9] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[10] Norman Ramsey. Literate programming simplified. *IEEE Software*, pages 97–105, Septemger 1994.

[11] Brian Selic, Garth Gullekson, and Paul T. Ward. *Real-time object-oriented modeling.* John Wiley & sons, 1994.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	12 April 1996	Final Report 18-Oct-95 to 12-Apr-96

4. TITLE AND SUBTITLE
Cooperating Formal Methods
Final Report

5. FUNDING NUMBERS
N00014-95-C-0349

6. AUTHOR(S)
David Guaspari

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Odyssey Research Associates, Inc.
301 Dates Drive
Ithaca, NY 14850-1326

8. PERFORMING ORGANIZATION REPORT NUMBER
ORA-TM-96-0019

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5660

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
A - unlimited release

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Formal methods research has developed a variety of mathematical tools and techniques applicable to the development of software systems, but they are greatly underused. The reasons include the limited mathematical backgrounds of many end-users and developers; idiosyncratic support tools; lack of attention to technology transfer. We have developed a prototype formal methods interface (PMI) that permits different kinds of users, with different degrees of expertise, to cooperate in applying formal methods to define, explore, and analyze system designs and specifications. Systems are specified in RSML, a mixed graphical and textual notation for defining state machines. The FMI allows a user to perform automated analysis of certain semantic questions about an RSML specification -- e.g., the question of whether the specified state machine can respond to every input. It does so by generating a representation of each question within the formalism of various formal analysis tools. The FMI can easily be linked to a variety of formal tools, as we have shown by rapidly incorporating the EVES and PVS theorem provers, and the SPIN model checker.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Formal methods, formal specification, automated theorem proving, model checking.	47
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	None